CS61A Notes – Week 07a:
Applicative and Normal Order, Lazy evaluator, Nondeterministic evaluator

Applicative vs. Normal Order

First, a few functions to play with:

```
(define (double x) (+ x x))
(define (square y) (* y y))
(define (f z) (+ (square (double z)) 1))
```

Scheme uses "applicative order", where it first completely evaluates all the procedures and arguments before executing the function call. Another way to do this – in "normal order" – is to expand out complex expressions involving defined procedures into one involving only primitive operators and self-evaluating values, and then perform the evaluation. In other words, we *defer* evaluation of the arguments; we substitute the *unevaluated* expression (as opposed to evaluated expression for applicative order) into the body of a procedure.

So for the problem `(f (+ 2 1))`:

- We'll expand `f` into its body first, leaving `(+ 2 1)` unevaluated:

`(+ (square (double (+ 2 1))) 1)`

- Now, we want to expand `square`; let's write in the body of `square`, and substitute `(double (+ 2 1))` for `y`, leaving `(double (+ 2 1))` unevaluated:

`(+` **`(* (double (+ 2 1)) (double (+ 2 1)))`** `1)`

- Now, we want to expand `double`; let's write the body of `double`, and substitute `(+ 2 1)` for `x`:

`(+ (*` **`(+ (+ 2 1) (+ 2 1)) (+ (+ 2 1) (+ 2 1))`**`) 1)`

- Finally we've expanded everything to just primitive operators and self-evaluating values. We'll do these one by one…

```
(+ (* (+ 3 (+ 2 1)) (+ (+ 2 1) (+ 2 1))) 1)
(+ (* (+ 3 3) (+ (+ 2 1) (+ 2 1))) 1)
(+ (* (+ 3 3) (+ 3 (+ 2 1))) 1)
(+ (* (+ 3 3) (+ 3 3)) 1)
(+ (* 6 (+ 3 3)) 1)
(+ (* 6 6) 1)
(+ 36 1)
37
```

Well okay, that seems harmless enough. It's just two ways of thinking about the problem: for applicative order, we evaluate completely all subexpressions first before we apply the current procedure. For normal order, we expand all the defined procedures into their bodies until everything is expressed in terms of primitive operations and self-evaluating values; then we find the answer.

So what's the difference? Some of you will raise concerns about efficiency (obviously, in this case, normal order causes us to perform `(+ 2 1)` four times, whereas we only performed the same calculation once for applicative order!). But in terms of correctness, the two are the same. That is, **in *functional programming*, applicative and normal order of evaluation will *always* return the same answer!**

QUESTIONS

1. **Above, applicative order was more efficient. Define a procedure where normal order is more efficient.**

2. **Evaluate this expression using both applicative and normal order: `(square (random x))`. Will you get the same result from both? Why or why not?**

3. **Consider a magical function `count` that takes in no arguments, and each time it is invoked, it returns 1 more than it did before, starting with 1. Therefore, `(+ (count) (count))` will return 3. Evaluate `(square (square (count)))` with both applicative and normal order; explain your result.**

The Lazy Way Out

The lazy evaluator implements the normal order of evaluation. We do this by delaying *execution as much as possible by* thunking *it; when we can't delay it further, we* force *the thunk to obtain the actual value.*

A **thunk** in the lazy evaluator is a list whose first element is the word `thunk`, second element the expression we're delaying, and third element the environment in which to evaluate the expression when we need to force the thunk.

There's only one place where we delay evaluation: **we thunk all arguments to a compound procedure**.

There are four places in which we want to force:
1. We always force the operator of a procedure call
2. We always force arguments to a primitive procedure
3. We always force what we're printing out to the screen on the top level
4. In special cases of special forms, like the predicate of `if`, we also want to force thunks

The procedure that actually does the delaying is called `delay-it`:

```
(define (delay-it exp env)
    (list 'thunk exp env))
```

Obviously, this just creates a thunk that looks just as we described above. The procedure that we use to force an expression is called `actual-value`:

```
(define (actual-value exp env)
    (force-it (mc-eval exp env)))
```

Note that `actual-value` *always* takes in a valid Scheme expression, and never a thunk! Thus, it can first call `mc-eval` to evaluate the expression. Then, since `mc-eval` might return a thunk, it needs to call `force-it` to actually force the thunk to get the real answer.

Note that the lazy evaluator also memoizes forced thunks. Take a look at the memoizing version of `force-it`:

```
(define (force-it obj)
    (cond ((thunk? obj)
           (let ((result (actual-value (thunk-exp obj) (thunk-env obj))))
               (set-car! obj 'evaluated-thunk)
               (set-car! (car obj) result)
               (set-cdr! (cdr obj) '())
               result))
          ((evaluated-thunk? obj)
           (thunk-value obj))
          (else obj)))
```

First, note the mutual recursion between `force-it` and `actual-value`; one calls the other. This is because forcing an object might yield another thunk, so we need to keep forcing it until we get some actual value. And note how we perform the memoization: when we find a thunk, we first call `actual-value` on the delayed expression to find the value. Then, we *transform* the thunk into an *evaluated-thunk*, and store the result of the forcing into the evaluated-thunk. From then on, if we try to force this evaluated-thunk again, we simply return the value we've already calculated.

QUESTIONS: What is printed at each line?

1.
```
> (define x (+ 2 3))
> x => ?
> (define y ((lambda (a) a) (* 3 4)))
> y => ?
> (define z ((lambda (b) (+ b 10)) y))
> z => ?
```

2.
```
> (define count 0)
> (define (foo x y) (x y))
> (define z (foo (lambda (a) (set! count a) (* a a))
                 (begin (set! count (+ 1 count)) count)))
> count => ?
> z => ?
```

3.
```
> (define count 0)
> (define (incr!) (set! count (+ count 1)))
> (define (foo x)
      (let ((y (begin (incr!) count)))
          (if (<= count 1)
              (foo y)
              x)))
> (foo 10) => ?
```

Nondeterministic and Indecisive

The nondeterministic evaluator extends the metacircular evaluator with nondeterministic searching. It's good, clean American fun. First we take a look at the new special form, amb*, which takes in any number of arguments. If there are no arguments, it* fails*. If there are arguments, it chooses the first one; if the first one causes a failure, it chooses the second one; if that one causes a failure, it chooses the third, etc., until it runs out of arguments, at which point it itself signals a failure.*

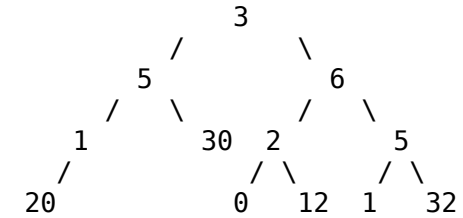
A companion to `amb` is `require`, which takes in a predicate value, and causes a failure if the value is `#f`:

```
(define (require pred?)
    (if (not pred?) (amb)))
```

Since you are trained on the imperative and functional ways of programming, this makes the structure of your program very unintuitive. But just imagine that the nondeterministic evaluator works magically; when an error is signaled by `(amb)`, you are allowed to fly to the nearest non-empty call to `amb`, try the next option and start over.

The book and the lectures already present several examples of programs written for the nondeterministic evaluator. But I will provide another motivating example…

Consider the problem of finding a path in a binary tree from the root to an element. Consider this tree `t`:



Note that this is not a binary search tree! Here's what we'd like:

```
> (path-to t 12)
(r l r) ;; to get from root to 12, go right, left, right
> (path-to t 32)
(r r r) ;; to get to 32, go right, right, right
> (path-to t 19)
#f ;; 19 is not in the tree
```

Here's how we'd need to write `path-to` in good old Scheme:

```
(define (path-to tree x)
    (cond ((empty-tree? tree) #f)
          ((eq? (datum tree) x) '())
          (else
              (let ((result (path-to (left-branch tree) x)))
                  (if result
                      (cons 'L result)
                      (let ((result (path-to (right-branch tree) x)))
                          (if result
                              (cons 'R result)
                              #f)))))))
```

An ugly little thing! But suppose you take advantage of the nondeterministic evaluator:

```
(define (path-to tree x)
    (cond ((empty-tree? tree) (amb))
          ((eq? (datum tree) x) '())
          (else (amb (cons 'L (path-to (left-branch tree) x))
                     (cons 'R (path-to (right-branch tree) x))))))
```

In the recursive case, we use `amb` to choose *either* walking down the left branch or the right branch of the tree. Suppose we chose left; then, if, eventually, we reach an empty tree, we're going to signal an error, causing us to switch to right. If that throws another error, we ourselves will signal an error (since our `amb` has run out of options), and our parent will explore other options (either trying the right branch or signaling failure to its parent).

Even better – if there are two elements in the tree, we can get the path to one and, after entering `try-again`, can get the path to the other. Think of the nightmare of doing that with regular Scheme!

QUESTIONS

1. **Suppose we type the following into the amb evaluator:**
```
> (* 2 (if (amb #t #f #t)
           (amb 3 4)
           5))
```
What are all possible answers we can get?

2. **Write a function `an-atom-of` that dispenses the atomic elements of a deep list (not including empty lists). For example,**
```
> (an-atom-of '((a) ((b (c)))) => a
> try-again => b
```

3. **Use `an-atom-of` to write `deep-member?`.**

4. **Fill in the blanks:**
```
> (define (choose-member L R)
      (cond ((null? R) (amb))
            ((= (car L) (car R)) (car L))
            (else (amb (choose-member L (cdr R))
                       (choose-member (cdr L) R)))))
> (choose-member '(1 2 3) '(4 2 3))

_____________________________
> try-again

_____________________________
> try-again

_____________________________
```